The Concept of Dynamic Analysis

Thomas Ball

Bell Laboratories
Lucent Technologies
tball@research.bell-labs.com

Abstract. Dynamic analysis is the analysis of the properties of a running program. In this paper, we explore two new dynamic analyses based on program profiling:

- *Frequency Spectrum Analysis.* We show how analyzing the frequencies of program entities in a single execution can help programmers to decompose a program, identify related computations, and find computations related to specific input and output characteristics of a program.
- *Coverage Concept Analysis.* Concept analysis of test coverage data computes dynamic analogs to static control flow relationships such as domination, postdomination, and regions. Comparison of these dynamically computed relationships to their static counterparts can point to areas of code requiring more testing and can aid programmers in understanding how a program and its test sets relate to one another.

1 Introduction

Dynamic analysis is the analysis of the properties of a running program. In contrast to static analysis, which examines a program's text to derive properties that hold for all executions, dynamic analysis derives properties that hold for one or more executions by examination of the running program (usually through program instrumentation [14]). While dynamic analysis cannot prove that a program satisfies a particular property, it can detect violations of properties as well as provide useful information to programmers about the behavior of their programs, as this paper will show.

The usefulness of dynamic analysis derives from two of its essential characteristics:

- *Precision of information*: dynamic analysis typically involves instrumenting a program to examine or record certain aspects of its run-time state. This instrumentation can be tuned to collect precisely the information needed to address a particular problem. For example, to analyze the shape of data structures created by a program (lists, trees, dags, etc.), an instrumentation tool can be created to record the linkages among heap-allocated storage cells.

- *Dependence on program inputs*: the very thing makes dynamic analysis incomplete also provides a powerful mechanism for relating program inputs and outputs to program behavior[15]. With dynamic analysis it is straightforward to relate changes in program inputs to changes in internal program behavior and program outputs, since all are directly observable and linked by the program execution. Viewed in this light, dynamic and static analysis might be better termed "input-centric" and "program-centric" analysis, respectively.

Dynamic and static analyses are complementary techniques in a number of dimensions:

- *Completeness.* In general, dynamic analyses generate "dynamic program invariants", properties which are true for the observed set of executions. [11] Static analysis may help determine or not these dynamic "invariants" truly are invariants over all program executions. In the cases where the dynamic and static analyses disagree, there are two possibilities: 1. the dynamic analysis is in error because it did not cover a sufficient number of executions; 2. the static analysis is in error because it analyzed infeasible paths (paths that can never execute). Since dynamic analysis examines actual program executions, it does not suffer from the problem of infeasible paths that can plague static analyses. On the other hand, dynamic analysis, by definition, considers fewer execution paths than static analysis.
- *Scope.* Because dynamic analysis examines one very long program path, it has the potential to discover semantic dependencies between program entities widely separated in the path (and in time). Static analysis typically is restricted in the scope of a program it can analyze effectively and efficiently, and may have trouble discovering such "dependencies at a distance".
- *Precision.* Dynamic analysis has the benefit of examining the concrete domain of program execution. Static analysis must abstract over this domain in order to ensure termination of the analysis, thus losing information from the start. Abstraction can be a useful technique for reducing the run-time overhead of dynamic analysis and reducing the amount of information recorded, but is not required for termination.

In this paper, we illustrate and discuss some of these concepts of dynamic analysis using program profiles [3]. A program profile counts the number of times program entities occur in a program execution. For example, a statement level profile counts how many times each statement executes. Profiles can be recorded at many different levels, from that of objects, methods and procedures, down to paths, branches and even individual machine instructions. Profiling tools are commonplace today, with most compilers and operating systems providing accompanying profiling toolsets.

We propose two new dynamic analyses based on program profiling:

- *Frequency Spectrum Analysis (FSA).* The idea behind FSA is that analyzing the frequencies of program entities in a single execution can help programmers to decompose a program, identify related computations, and find

computations related to specific input and output characteristics of the program. We demonstrate FSA on a small obfuscated C program that prints the poem "The Twelve Days of Christmas". For this case study, we used path profiling [1] technology to monitor the execution behavior of the program. Based on our analysis, we created an "unobfuscated" version of the program that retains the original program's profile signature and clearly explains the operation of the original program.
- *Coverage Concept Analysis (CCA).* We show how concept analysis applied to coverage profiles naturally computes dynamic analogs to static control flow relationships such as domination and regions, identifying "dynamic control flow invariants" across a set of executions. Comparison of the dynamically invariant control flow relationships to their static counterparts can point to areas of code requiring more testing and can aid programmers in understanding how their code and test sets relate to one another.

This paper is organized as follows. Section 2 presents the basic ideas behind frequency spectrum analysis and our case study of the obfuscated C program. Section 3 reviews concept analysis and shows the different ways in which it can help us to understand the relationships between tests and coverage information. Section 4 discusses related work. Section 5 concludes the paper.

2 Frequency Spectrum Analysis

This section presents the ideas behind frequency spectrum analysis (FSA) and then describes how this analysis was used to help understand the internal behavior of an obfuscated C program.

2.1 The Meaning of Frequencies

The traditional use of program profiles in performance tuning is to separate the frequently executed parts of a program from the less frequently parts. By delving a bit deeper into the information in program profiles (that is, the frequencies of the program entities, as recorded in a profile), FSA can help a programmer in three basic tasks:

- partitioning the program by levels of abstraction;
- finding related computations;
- find computations related to specific attributes of a program's input or output.

In the next section, we will present our analysis of an obfuscated C program based on several general observations made in this section. Table 1 shows the path profile of the obfuscated C programs' execution (Figure 1). Twelve paths executed and each path's static identifier (composed of the procedure name containing the path and the path's integer identifier in that procedure) and execution frequency are shown. The paths are sorted in ascending order of

Path ID	Frequency	Path ID	Frequency
main:0	1	main:2	114
main:19	1	main:3	114
main:22	1	main:1	2358
main:23	10	main:7	2358
main:9	11	main:4	24931
main:13	55	main:5	39652

Table 1. A path profile of the (readable) obfuscated C program's execution.

frequency. We will use this path profile to motivate FSA, without reference to the program's output or its code. In the next section, we will analyze how the paths and frequencies are related to the program's output and structure.

FSA is based on three simple observations about how frequencies relate to program behavior:

- *Low Versus High Frequencies.* The relative execution frequencies of program entities can provide clues as to their place in the hierarchy of program abstractions. For example, the interface procedures to a sorting module generally will be called many fewer times than the private procedures in the module that invoke one another to perform the sort operation. In object-oriented programs, methods implementing a high-level architectural pattern probably will have lower execution frequency than methods implementing the guts of an algorithm.

  In Figure 1, we immediately see that the paths main:4 and main:5 have much higher frequencies than the other ten paths. This indicates that these paths are involved in some highly repetitive computation.
- *Related Frequencies and Frequency Clusters.* The fact that a procedure foo is called 1033 times may not be particularly noteworthy. However, the fact that procedures foo and bar each are called 1033 times usually is more than mere coincidence. This is the basic idea behind related frequencies or "frequency clusters".

  The reason for such frequency clustering may be that procedure foo always calls procedure bar, or that there is another procedure foobar that calls both foo and bar. There can be many explanations for a frequency cluster. Regardless of the underlying mechanism that created the cluster, the cluster by itself is an interesting hint to the programmer about dynamic relationships between program entities that may not be apparent in the static program structure. Frequency clusters partition the program many ways, slicing across traditional abstraction boundaries, as entities widely separated in program text may be related to one another through common frequency.

  Two clusters are immediately apparent in the path profile of Figure 1: paths main:2 and main:3 with frequency 114 and paths main:1 and main:7 with frequency 2358.

```
#include <stdio.h>
main(t,_,a)char*a;{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Fig. 1. An obfuscated C program to print the poem "The Twelve Days of Christmas". The partial output of the program is shown in Figure 5.

- *Specific Frequencies.* Knowledge about the characteristics of program's input or output can greatly aid in FSA. For example, if the output of a program is an enumeration of records, there is probably a program entity whose frequency is the size of this enumeration. Frequencies related to the input or output domain of a program can help a programmer identify those parts of a program responsible for input or output. This idea can be extended in several obvious directions. For example, one can look for frequencies that might indicate a $O(N^2)$ algorithm, as suggested by [19].

As suggested above, profiles contain a wealth of information that is rarely exploited by programmers. Jon Bentley, in his series of columns and books on writing efficient programs, discusses how execution counts "tell interesting tales" and can help programmers to debug misbehaving programs as well as to tune the performance of well behaved programs. [5, 6] In the next section, we explore this idea in some detail through a case study.

2.2 Case Study: Understanding an Obfuscated C Program

Figure 1 presents an obfuscated C program that often makes the rounds during the holiday season (the author has received it at least twice). The program takes no input[1]and produces the poem "The Twelve Days of Christmas", an excerpt of which is presented in Figure 5 in the Appendix.

In this section, we will show how we used FSA to help determine how the program accomplishes the printing of the poem and to create a new "unobfus-

[1] It should be noted that in this very special circumstance, a dynamic analysis is a static analysis. Nonetheless, the information compute by the dynamic analysis (profiles) is unavailable from conventional static analyses.

```
#include <stdio.h>
main(t,_,a) char *a;
{
     if ((!0) < t) {
[1]    if (t < 3)  main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a));
[2]    if (t < _ ) main(t+1,_,a);
[3]    main(-94,-27+t,a);
[4]    if (t==2 && _ < 13 ) main(2,_+1,"");
     } else if (t < 0) {
[5]    if (t < -72)  main(_,t,LARGE_STRING);
       else if (t < -50 ) {
[6]      if (_ == *a) putchar(31[a]);
[7]      else         main(-65,_,a+1);
[8]    } else main((*a=='/')+t,_,a+1);
[9]  } else if (0 < t) main (2,2,"%s");
[10] else if (*a!='/') main(0,main(-61,*a,SMALL_STRING),a+1);
}
```

Fig. 2. A (more) readable version of the obfuscated C program, after reformatting, performing local syntactic substitutions to turn expressions into statements and eliminating dead code. There are 10 lines containing calls, each uniquely numbered in brackets.

cated" program that explains how the original program works. In restructuring the program, we maintained as much of the original program's computational signature as possible. Whenever possible, we rewrote the program in the spirit of the original program, rather than substituting a radically different piece of code in place of one we didn't happen to like.

Making the Program Readable To understand a program, it first is helpful to be able to read it. The given program is barely readable, even for those very familiar with the C language. Our first task was to reformat the code, using indentation and explicit parenthesization to make it more readable, as well as rewriting it without the use of conditional or list expressions. Figure 2 shows the result of these local syntactic transformations.

The readable obfuscated program consists of one function `main` with three arguments (`t`, `_` and `a`) and calls itself repeatedly. The second argument is an underscore, which is a legal variable name in C. The function `main` truly is a function, as it does not update any variables. It achieves its goal based solely on the values passed to it. The initial invocation of the program will cause the value of parameter `t` to be 1 (because in Unix, the first argument to `main` is the count of the number of arguments on the command line including the name of the program itself). The program contains two strings (shown in the original program in Figure 1, but elided here to `LARGE_STRING` and `SMALL_STRING`, which appear to encode the text of the poem.

Path ID	**Frequency**	**Condition**	**Call Lines**
main:0	1	t == 1	[9]
main:19	1	t==2 && t >= _	[1,3,4]
main:22	1	t==2 && t < _ && _ >= 13	[1,2,3]
main:23	10	t==2 && t < _ && _ < 13	[1,2,3,4]
main:9	11	t >= 3 && t >= _	[3]
main:13	55	t >= 3 && t < _	[2,3]
main:2	114	t == 0 && *a == '/'	*no call lines*
main:3	114	t < -72	[5]
main:1	2358	t == 0 && *a != '/'	[10]
main:7	2358	t > -72 && t < -50 && _ == *a	[6]
main:4	24931	t < 0 && t >= -50	[8]
main:5	39652	t > -72 && t < -50 && _ != *a	[7]

Table 2. Summary of the twelve executed paths in the readable obfuscated C program of Figure 2.

The Frequency Spectrum Analysis Before taking on a reverse engineering task, it is important to have some model in mind to help guide the process. The "Twelve Days of Christmas" is all about counting gifts, so we approach the poem and the program by identifying various quantities that arise from the poem's natural structure:

- *12* verses, on for each of the 12 days of Christmas.
- *26* unique strings: there are many repeated strings in the poem. There are three strings for the common structure ("On the", "day of Christmas...", "and a partridge ..."), 12 strings for the ordinals, and 11 strings for the second through twelfth gifts, giving a total of 26 unique strings.
- *66* occurrences of presents other than a "partridge in a pear tree" (which occurs in every verse).
- *114* strings printed: 12 occurrences of the three common strings (36), 12 ordinals, and 66 non-partridge gifts (36 + 12 + 66 = 114);
- *2358* characters printed as output, as counted by the Unix word count utility `wc`.

We have seen some of these frequencies before in Figure 1. Recall that the goal of FSA is to use the frequencies obtained from a program profile to aid in understanding the program. The idea is that these execution counts will help us identify which parts of the program are responsible for which parts of the poem. For example, a program element with an execution count of 11 or 12 may indicate an entity involved in the control of the number of verses, while an element with an execution count of 2358 is most likely involved in printing characters.

We used the PP path profiling tool of Ammons, Ball and Larus [4, 1] to capture intraprocedural path[2]execution counts of the readable program. The program takes no input, so there is only one path profile to consider. Table 2 repeats the twelve executed paths in the path profile of the readable program from Table 1, with some additional information. For this program, each path is uniquely identified by the conditions on the parameters `t`, `_` and `a` and by the lines in the path that contain procedure calls (referred to here as "call lines"). There are ten lines containing procedure calls in the code in Figure 2, labelled in brackets. The path condition and the procedure call lines in each path are summarized in Table 2.

The first thing that is apparent from Table 2 is that there is a strong correlation between a path's frequency and the call lines that it covers. Paths with frequencies less than 100 cover subsets of call lines in the set { 1,2,3,4,9 }, while each path with frequency greater than 100 covers a different call line not in this set. A closer examination of the code and the paths shows that the paths cluster into six main groups (separated by the double lines in the table), as detailed below:

- Path main:0 (executed once) initializes the recursion.
- Paths main:19, main:22, and main:23 control the printing of the 12 verses. In particular, path main:19 represents the first verse, path main:23 the middle 10 verses, and path main:22 the last verse. The sum of these paths' frequencies is 12, the number of verses in the poem. Each of the paths covers a different set of recursive calls to `main` (call lines 1-4). These paths helped us identify that certain calls were responsible for the first line of each verse (call line 1), starting the inner loop iteration to print the list of gifts (call line 2). printing a single gift (call line 3), as well as iterating the outer loop (call line 4).
- Paths main:9 and main:13 control the printing of the non-partridge-gifts within a verse. Note that the frequencies of the two paths sum to 66, as expected from our analysis of the poem. These paths make up the "inner loop" of the program.
- Paths main:2 and main:3 are responsible for printing out a string. Each path has frequency 114, the exact number of strings predicted by analyzing the poem's structure. The path main:3 represents the initialization (passing the large string in as parameter `a`) and the path main:2 represents the termination of the printing of the string (when the '/' separator is found).
- Paths main:1 and main:7 print out the characters in a string. Each path executes 2358 times. Why are there two paths with frequency 2358? We will soon see.
- What about the anomalous paths main:4 and main:5 with the large frequencies of 24931 and 39652? Examination of the code reveals that path main:4 is responsible for skipping over `t` sub-strings in `LARGE_STRING` to get

[2] Intraprocedural paths do not follow control flow from a call site to the entry of the called procedure. They stay in the same procedure (effectively treating the procedure call as if it had no effect on the control flow).

to the $t+1^{th}$ sub-string. Each sub-string is terminated with the '/' character. Every time the $t+1^{th}$ sub-string is to be printed, a linear scan through the large string is done to get to that sub-string, which accounts for path main:4's high frequency.
Path main:5 scans `SMALL_STRING` until it finds the character in it that matches the current character (the value of the argument `_`) to be printed, at which point path main:7 executes. The character 31 positions later in the small string (`31[a]`, which in C is equivalent to `a[31]`) is the translation of the character. This explains why there are two paths with frequency 2358. Path main:1 is the initiation of the search of the small string to find the character translation and path main:7 performs the translation and printing of the character. Path main:5's high frequency is due to the fact that the small string is scanned each time for every character to be printed.

The Restructured Program Using the knowledge gained from FSA and manual examination of the program, we restructured the program to produce the program shown in Figure 3. We strove to keep the recursive structure of the program intact, but used different functions to represent the different tasks of the original program, as captured by the clustering of the paths. We did not change the values of the two relevant text strings (the list of sub-strings of the poem, `LARGE_STRING`, and the translation mapping, `SMALL_STRING`). The original program used the value 2 to represent the first day of Christmas. We shifted this down to 1 to match the poem.

There are seven functions in the new program, corresponding closely to the clusters of paths identified in the old program:

- `main` (path main:0);
- `outer_loop` (paths main:19, main:22 and main:23);
- `inner_loop` (paths main:9 and main:13);
- `print_string` (paths main:2 and main:3);
- `output_chars` (paths main:1 and main:7) and `translate_and_put_char` (path main:5);
- `skip_n_strings` (path main:4).

The new program has the exact same output as the old, and all of the performance disadvantages as well. To show that we have (in some sense) captured the essence of the original program, we path profiled the new program. The path profile of the new program is shown in Table 3, with paths sorted in ascending order of frequency; it is very similar to the original profile (Table 2) with some minor differences due to the restructuring.

Summary A well known folk theorem in computer science is that any program can be transformed into a semantically equivalent program consisting of a single recursive function. This is what makes the obfuscated "12 Days of Christmas" program most difficult to understand. The first parameter to the function `main`

```
#include <stdio.h>
static char *strings = LARGE_STRING;     /* the original set of strings */
static char *translate = SMALL_STRING;     /* the translation mapping */
#define FIRST_DAY  1
#define LAST_DAY  12

/* the original "indices" of the various strings */
enum { ON_THE = 0, FIRST = -1, TWELFTH = -12, DAY_OF_CHRISTMAS = -13,
       TWELVE_DRUMMERS_DRUMMING = -14, PARTRIDGE_IN_A_PEAR_TREE = -25
};

char* skip_n_strings(int n,char *s) { /* skip -n strings (separator is /), */
  if (n == 0)  return s;               /* where n is a negative value */
  if (*s=='/') return skip_n_strings(n+1,s+1);
  else         return skip_n_strings(n,s+1);
}

/* find the character in the translation buffer
   matching c and output the translation */
void translate_and_put_char(char c, char *trans) {
  if (c == *trans) putchar(trans[31]);
  else             translate_and_put_char(c,trans+1);
}

void output_chars(char *s) {
  if (*s == '/') return;
  translate_and_put_char(*s,translate);
  output_chars(s+1);
}

/* skip to the "n^th" string and print it */
void print_string(int n) {  output_chars(skip_n_strings(n,strings)); }

/* print the list of gifts */
void inner_loop(int count_day, int current_day) {
  if (count_day < current_day) inner_loop(count_day+1,current_day);
  print_string(PARTRIDGE_IN_A_PEAR_TREE+(count_day-1));
}

void outer_loop(int current_day) {
  print_string(ON_THE);               /* "On the " */
  print_string(-current_day);         /* ordinal, ranges from -1 to -12 */
  print_string(DAY_OF_CHRISTMAS);     /* "day of Christmas ..." */
  inner_loop(FIRST_DAY,current_day); /* print the list of gifts */
  if (current_day < LAST_DAY)
    outer_loop(current_day+1);
}

void main() { outer_loop(FIRST_DAY); }
```

Fig. 3. The restructured "The Twelve Days of Christmas" program.

Path ID	Frequency	Path ID	Frequency
main:0	1	skip_n_strings:0	114
outer_loop:0	1	skip_n_strings:2	1898
outer_loop:1	11	output_chars:0	2358
inner_loop:0	12	translate_and_put_char:2	2358
inner_loop:1	66	skip_n_strings:1	23033
output_chars:1	114	translate_and_put_char:0	39652
print_string:0	114		

Table 3. The path profile of the restructured program.

takes on the role of the program counter and parameters are overloaded to have different interpretations depending on the context they are used.

We used FSA to help separate out the set of functions that this single function implements. Thus small case study illustrates the essential features of FSA:

- The use of low versus high frequencies to partition the program by levels of abstraction (for example, the printing of verses as compared to scanning of strings);
- The use of frequency clusters to identify related computations in the program (for example, the paths comprising the outer and inner loops);
- The use of specific frequencies to find computations related to the program's observed behavior (for example, the paths responsible for printing a substring or a character).

Our analysis clearly leaves many questions unanswered. Although complex, the obfuscated C program was quite small. How will FSA scale to larger programs with accompanying larger profiles? There are a number of issues here. With the obfuscated C program, there was a rather direct relationship between attributes of the program's output and the program's behavior. With larger programs containing complex intermediate computations, we cannot hope to find such direct relationships. The size of the profile is also an issue, as there will generally be a lot of "noisy" data surrounding the data that one is interested in. We feel that the three basic observations of FSA (low vs. high frequency, frequency clusters, and special frequencies) will continue to be useful for larger programs, but only experience will show how.

Another shortcoming of our case study was that the obfuscated C program had no inputs. The appearance of the same frequency correlations across different executions (even if absolute frequency values are different) would provide stronger evidence of semantic relationships between parts of a program. In the next section, we discuss an approach to help analyze multiple execution profiles and compare the relationships in program executions to their static counterparts in program source text.

3 Coverage Concept Analysis

The previous section demonstrated how analysis of the frequency spectrum of a single program execution can help in understanding and decomposing a program. What can be done if there are many executions to be examined? This section considers this question for a restricted but very commonly used type of profile, the coverage profile, which records for each test run, the entities that executed (but not their frequencies).

The main result of this section is to show that concept analysis applied to coverage profiles naturally computes dynamic analogs to static control flow relationships such as domination and regions, identifying "dynamic control flow invariants" across a set of executions. Additionally, the comparison of the dynamically invariant control flow relationships to their static counterparts can point to areas of code requiring more testing and can aid programmers in understanding how their code and test sets relate to one another.

3.1 Concept Analysis and Test Coverage

Concept analysis is a technique for identifying groups of objects that have common attributes [10]. The input to concept analysis is a binary relation between objects and attributes. This relation can be represented as a boolean-valued table in which rows represent objects and columns represent attributes. An entry of the table is true if an object has an attribute and false otherwise.

For our purposes, the objects (rows) are tests and the attributes (columns) are the program entities that a test may cover, such as the procedures, statements, branches or paths of the program. Figure 4(a) shows an example of a test coverage table that can be input to concept analysis. The table shows procedure-level coverage of five tests (t1 through t5) of an implementation of a red-black tree data structure (a form of balanced binary tree). The procedure names have been shortened to make the table more compact.

In the testing domain, the pair (T, E), where T is a set of tests and E a set of program entities, is a *concept* if every test in T covers all the entities in E, and no test outside of T covers all the entities in E. Equivalently, (T, E) is a concept if every entity in E is covered by every test in T and there is no entity outside of E covered by every test in T. Stated yet another way, concepts determine maximal sets of tests covering identical entities (and maximal sets of entities covered by identical tests). Concepts can be computed by a variety of algorithms [12, 18]. In the worst-case, for a table of size n rows by n columns, there may be 2^n concepts, so the worst-case running time of any batch algorithm that computes all concepts is exponential in n. In practice, concept lattices have $O(n^2)$ concepts and sometimes even $O(n)$ concepts [18].

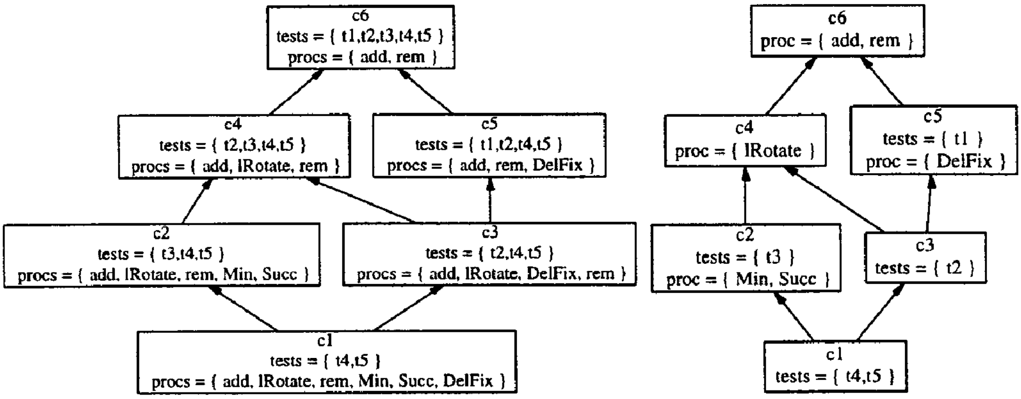
The table in Figure 4(a) gives rise to six concepts, shown in Figure 4(b). The concept c4 has the tests { t2,t3,t4,t5 }, which have the procedures { `add, rem, lRotate` } in common. Furthermore, this set of procedures has exactly the tests { t2,t3,t4,t5 } in common. The pair ({ t1, t2 }, { `add, rem, DelFix` }) is not a

	Procedures					
Test	add	lRotate	rem	Min	Succ	DelFix
t1	X		X			X
t2	X	X	X			X
t3	X	X	X	X	X	
t4	X	X	X	X	X	X
t5	X	X	X	X	X	X

(a)

Concept	Tests	Procedures
c1	t4, t5	add, lRotate, rem, Min, Succ, DelFix
c2	t3, t4, t5	add, lRotate, rem, Min, Succ
c3	t2, t4, t5	add, lRotate, rem, DelFix
c4	t2, t3, t4, t5	add, lRotate, rem
c5	t1, t2, t4, t5	add, rem, DelFix
c6	t1, t2, t3, t4, t5	add, rem

(b)



(c) (d)

Fig. 4. (a) Partial procedure coverage from five tests of a red-black tree implementation; (b) The six concepts this coverage information induces; (c) Concept lattice of with full labelling of tests and procedures; (d) Concept lattice with minimal labelling.

concept because the set { t1, t2 } is not the maximal set of tests with common entities { `add, rem, DelFix` } (as concept c5 illustrates).

Concepts can be ordered by set inclusion on tests or entities. The set of all concepts forms a complete partial order ($\sqsubseteq$), given by:

$$(T_1, E_1) \sqsubseteq (T_2, E_2) \iff T_1 \subseteq T_2 \iff E_2 \subseteq E_1$$

This partial order is also referred to as the *concept lattice.* Figure 4(c) shows the concept lattice for the six concepts c1 through c6, with one node for each concept. If $c \sqsubseteq d$ (and there is no concept c' such that $c \sqsubseteq c' \sqsubseteq d$) then there is an arrow $c \dashrightarrow d$ in the lattice. Each concept is labelled with its associated set of tests and set of entities.

There are a number of important properties of the concept lattice:

- If a test t is in a concept c then it is in any concept greater than c (higher in the lattice). Furthermore, if an entity e is in a concept c, then it is in any lesser concept (lower in the lattice). In the running example, test t3 is in concept c2 and so is also in concepts c4 and c6. Procedure `lRotate` is in concept c4, so it is also in concepts c1, c2, and c3.
- For every test t, there is a unique least concept in which it appears, denoted by *lcon*(t). Similarly, for every entity e, there is a unique greatest concept in which it appears, denoted by *gcon*(e). Concept c2 is the least concept containing test t3. Similarly, c4 is the greatest concept containing the procedure `lRotate`.

Figure 4(d) shows how the concept lattice can be labelled so that each test and entity appears exactly once. A concept c is labelled with a test t if and only if $c = lcon(t)$. Likewise, a concept c is labelled with an entity e if and only if $c = gcon(e)$. From now on, the term *concept lattice* is used to refer to the concept lattice labelled in this fashion. All the information in the input table can be recovered from this concept lattice.

3.2 Concepts and Control Flow Invariance

This section shows how concept analysis of the test-vs-entities table provides dynamic analogs to static control flow relationships such as domination, postdomination and regions. Concept analysis of tests-vs-entities identifies "dynamic control flow invariants" between entities over a set of tests. These "invariants" are dynamic because they not guaranteed to hold for all executions, but do hold for the set of observed executions (tests). The comparison of the dynamic and static control flow invariants in a program can be used to help develop new tests.

Domination, Postdomination, and Control Flow Implication Domination and postdomination are binary relations over the control flow entities of a program that identify when the execution of one entity implies the execution of another. Consider control flow entities e and f. Entity e is said to dominate entity f if every path from program entry to f includes e. Entity f is said to postdominate entity e if every path from e to program exit includes entity f.

If entity f dominates entity e then any test that covers e must also cover f. If f postdominates e then it is also the case that any test that covers e must also cover f. Thus, execution of entity e (statically) *implies* the execution of entity f if f dominates e or f postdominates e.

The partial ordering of concepts in the concept lattice provides the execution-time equivalent of control flow implication. If entity f is in a concept greater than or equal to $gcon(e)$ then the execution of e dynamically implies the execution of f. That is, whenever a test t covers e it also covers f. For example, consider the procedure `Min`, which labels concept c2 in Figure 4(d). Concept c4, which contains procedure `lRotate`, is greater than c2, so any test that covers `Min` also covers `lRotate`. However, as the lattice also shows, there is a test (t2) in which `lRotate` executes but `Min` does not.

Regions If the execution of entity e implies the execution of f and the execution of entity f implies the execution of e then e and f are said to occupy the same control flow *region*. That is, there is no test that can separate the execution of e from f. The entities either execute together or not at all. Regions partition the set of control flow entities in a program using the static domination and postdomination relations defined above. More precisely, entities e and f are in the same region if e dominates f and f postdominates e.[3]

As with control flow implication, the concept lattice also identifies entities that always execute together in a set of tests. If $gcon(e) = gcon(f)$ then e and f always execute together in the set of given tests. That is, they are in the same *dynamic region*. For example, in the concept lattice in Figure 4(d), procedures `Min` and `Succ` have the same greatest concept (c2), and thus always execute together. Also, the procedures `add` and `rem` share the concept c6. No other two procedures occupy a dynamic region.

Comparing Dynamic and Static Information This section shows how the comparison of the static and dynamic control flow relations defined in the previous sections can be a useful aid in the development of new tests.

Suppose a program has been run on a set of tests and there is a pair of elements e and f such that e dynamically implies the execution of f, yet e does not statically imply f's execution. Or suppose that $gcon(e) = gcon(f)$, yet e and f are in different static regions. There may be a test that covers entity e but does not cover entity f. On the other hand if the execution of e statically implies the execution of f or e and f occupy the same static region, there is no point in trying to find a test that covers e but does not cover f. In the example of Figure 4(d), the procedures `Min` and `Succ` always execute together. However, these procedures are in different static regions in the red-black tree program. In fact, there is a test that separates their execution.

[3] This is a particular type of region known as *weak regions*.[2] *Strong regions* identify code that will always be executed the same number of times.

This example shows how concept analysis provides an intermediate point between "entity-based" and "path-based" coverage criteria. Entity-based coverage criteria such a statement or branch coverage consider coverage of entities in isolation. Path-based coverage criteria consider sequences of entities (and so subsumes entity-based criteria), but infeasible paths greatly complicate determining what a sufficient level of coverage is. Concept analysis identifies entities that always execute together in a given set of tests (or whose execution implies the execution of other entities). By comparing this "set-based" coverage information to the static regions of a program, a programmer can determine those entities whose execution they might try to separate.

4 Related Work

Recent work on dynamic discovery of program invariants is closely related to our work [11]. Ersnt et al.'s work instruments program to record the values that variables take on in one or more executions. This information is input into an "invariant detection engine" the checks for a number of invariants, such as that a variable has a constant value or takes on a small number of values; or that a variables value is bounded by some range, etc. Restated, they discover logical invariants over a set of program executions, where the types of logical invariants that can be identified is another input to the analysis engine.

Frequency spectrum analysis identifies control flow invariants within an execution (such as that two entities execute the same number of times), while concept concept analysis of test coverage information identifies control flow invariants in a set of program executions. Some control flow invariants may imply some of the invariants that Ernst et al's machinery discovers, and vice versa. For example, if a control flow branch based on "x == 2" always evaluates true, then the control flow information implies that the variable x always has the value 2 at that point in the program. Value invariance and control flow invariance and techniques used to discover them are thus quite complementary.

Other work on using dynamic analysis for exploring program executions concentrates on "dynamic differencing" [20, 15]. The idea is very simple. Each execution of a program generates a different "profile spectrum", a different set of entities that are covered. This set is, of course, dependent on the input that a program reads and its interactions with the environment. By carefully controlling the inputs to a program and/or the environment in which it executes, perturbing these slightly and observing the differences in the sets of covered entities, one can determine which parts of the code are affected by the perturbations. Wilde proposed this technique as a way to determine which code in a telephone call processing system is responsible for different call features (such as Caller ID, Call Waiting, etc). In this case, different call scenarios would be used to generate the different profile spectrums, but with slight modifications to the set of calling features that were enabled. Reps et al. showed how dynamic differencing could be used to find code that is dependent on dates by simply changing those parts of the input to a program related to dates (i.e., years). Both Wilde's and Reps et al.'s techniques are based mainly on program coverage. Reps et al. also proposed using frequency information (counts rather than coverage) to refine the analysis.

Concept analysis of test-vs-coverage information is dynamic differencing of test coverage taken to the extreme. Concept analysis provides a full factoring of the coverage information that exposes not only the differences between tests, but what they have in common as well. In addition, it computes a number of useful relations, such as control flow correlation and test subsumption, in a single framework.

Much research has been done on applying concept analysis to aide in the understanding and restructuring of programs [18, 16, 17]. All such work that we are aware of applies the concept analysis machinery to static relationships in a program, such a "procedure P uses variable V", "class D inherits from class C", etc. Our work makes use of the same machinery, but applies it to the dynamic relationship of "test T covers entity E" in order to help understanding the execution behavior of programs across a set of tests.

A general idea behind frequency spectrum analysis is to use the dynamic behavior of programs to help construct models of their behavior. This basic idea has been explored in many related settings. For example, in the area of formal methods, many techniques for finite state machine synthesis have been proposed for constructing finite state models from a set of traces of observed program behavior [7]. Cook and Wolfe used such techniques for reverse engineering software processes [8] and later used related techniques to develop models from the traces of multi-process programs [9]. In the arena of object-oriented programs, a number of efforts have explored how to bridge the gap between programmer models of OO behavior and what happens in OO program execution [13]. These efforts typically instrument an OO program to record message sends and other information, and then use a GUI to help programmers understand the traces and build models from them.

5 Conclusions

We have shown how frequency spectrum analysis and concept analysis of program profiles can aid in the tasks of program comprehension, program restructuring, and new test development. Just as program databases about static program structure have aided programmers and testers in their jobs, databases of dynamic program behavior gathered over the history of a program should provide valuable the software production cycle. The questions of what dynamic data can be collected and stored and what tasks this data and analysis of it can support are matters for future investigation.

Acknowledgements

Thanks to Jim Larus for his comments on earlier drafts of this paper.

References

1. G. Ammons, T. Ball, and J.R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, June 1997. Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation.
2. T. Ball. What's in a region? or computing control dependences in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1-4):1–16, December 1993.
3. T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
4. T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of MICRO 96*, pages 46–57, December 1996.
5. J. Bentley. *Writing Efficient Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
6. J. Bentley. *Programming Pearls.* Addison-Wesley, Reading, MA, 1986.
7. A. W. Biermann and J. A. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, June 1972.
8. J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, July 1998.
9. J. E. Cook and A. L. Wolf. Event-based detection of concurrency. In *Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 35–45, November 1998.
10. B.A. Davey and H.A. Priestley. *Introduction to lattices and order.* Cambridge University Press, 1990.
11. M. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *21st International Conference on Software Engineering*, pages 213,224, Los Angeles, CA, May 1999.
12. R. Godin and R. Missaoui H. Alaoui. Incremental concept formation algorithms based on Galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, 1995.
13. Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of the 19th International Conference on Software Engineering*, pages 360–370, May 1997.
14. J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Software-Practice and Experience*, 24(2):197–218, February 1994.
15. T. Reps, T. Ball, M. Das, and J.R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering (Lecture Notes in Computer Science)*, Zurich, Switzerland, September 1997. Springer-Verlag.
16. Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *International Conference on Software Maintenance*, pages 170–179, Bari, Italy, October 1997.
17. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. In *Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 99–110, November 1998.

18. Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.
19. O. Waddell and J. M. Ashley. Visualizing the performance of higher-order programs. *Proceedings of the 1st Workshop on Program Analysis for Software Tools and Engineering (ACM SIGPLAN Notices)*, 33(7):75–82, July 1998.
20. Norman Wilde. Faster reuse and maintenance using software reconnaissance. Technical Report SERC-TR-75F, Software Engineering Research Center, CSE-301, University of Florida, CIS Department, Gainesville, FL, July 1994.

Appendix

```
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

...

On the twelfth day of Christmas my true love gave to me
twelve drummers drumming, eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.
```

Fig. 5. Partial output of the obfuscated C program.